Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation

Rijnard van Tonder
rvantonder@ml.sun.ac.za
MIH Media Lab, Stellenbosch University

Herman Engelbrecht
hebrecht@ml.sun.ac.za
MIH Media Lab, Stellenbosch University

Abstract

Increased focus on the Universal Serial Bus (USB) attack surface of devices has recently resulted in a number of new vulnerabilities. Much of this advance has been aided by the advent of hardware-based USB emulation techniques. However, existing tools and methods are far from ideal, requiring a significant investment of time, money, and effort. In this work, we present a USB testing framework that improves significantly over existing methods in providing a cost-effective and flexible way to read and modify USB communication. Amongst other benefits, the framework enables man-in-the-middle fuzz testing between a host and peripheral. We achieve this by performing two-way emulation using inexpensive bespoke USB testing hardware, thereby delivering capabilities of a USB analyzer at a tenth of the cost. Mutation fuzzing is applied during live communication between a host and peripheral, yielding new security-relevant bugs. Lastly, we comment on the potential of the framework to improve current exploitation techniques on the USB channel.

Keywords: USB, emulation, fuzzing, man-in-the-middle

1 Introduction

Exploration of the USB attack surface has drawn attention among security researchers as far back as 2005 [7], and while USB bugs and exploits have been sporadically reported since 2009 [14], it is only in the last two years that considerable headway has been made. This is evidenced by a slew of new vulnerability reports in 2013, registering far more USB security advisories than previous years combined.[1] These bugs are exhibited mostly in USB drivers of operating systems, and often lead to memory corruption vulnerabilities.

Perhaps the primary reason for the discovery of recent bugs is the maturity of cost-effective USB hardware which enables the injection of input into the USB channel [8]. One such piece of hardware, the Facedancer [2], is a bespoke USB testing device that emulates a USB device with the aid of software. This approach lends itself well to finding bugs throughout the USB software stack, including core drivers, third-party drivers, and application software which handle USB data. Dedicated USB equipment, such as USB analyzers, are arguably the most effective for this task, but often prove to be prohibitively expensive. Other viable approaches for testing specific USB software without the use of hardware have also been demonstrated [6]. Here, fuzz testing (or fuzzing) is made possible by emulating the presence of USB devices in a guest operating system. Unfortunately, this carries the limitation that USB software has to function within the constraints of the virtual operating system. Because the USB protocol interacts closely with hardware devices, this makes it difficult to achieve full USB functionality.

We therefore argue that a hardware-based approach should be preferred when exploring the USB attack surface. The cost-effectiveness and flexibility of a hardware-based approach, however, affect our ability to explore it efficiently. In practice, it is difficult to achieve both these goals with currently available solutions, which are discussed in depth in Section 5.

This paper presents a novel approach for finding bugs in USB software that is both cost-effective and highly flexible. The goal of the design is to allow one to tap into USB communication between a host and peripheral device. The data should not only be *readable* (affording functionality analogous to a bus-pirate device), but also *modifiable*, as typically made possible by a USB analyzer. Moreover, this is achieved with two bespoke USB emulation devices which establish two-way communication that is transparent to the host and peripheral. The attributes of the Transparent Two-Way Emulation (TTWE) framework thus enable us to perform man-in-the-middle fuzzing. Beyond supplying USB analyzer capabilities at a tenth of the cost, the framework also allows full control over the software that handles USB communication, unlike traditional bundled USB analyzer software. While the framework makes use of affordable hardware, it is a hybrid approach, and relies on a software component. This software component enables the desired flexibility and efficiency when exploring the USB attack surface, since its role is to transparently mediate USB communication between a host and peripheral.

In this paper we discuss the architecture of the framework, demonstrate the practical application thereof, and show that it is effective at finding USB bugs that may have security implications. The contributions of this work in particular are:

1. **A USB testing framework** that
   (a) Is *flexible*, by allowing man-in-the-middle modification of USB host/peripheral communication through software,
   (b) Is *cost-effective* in affording USB analyzer functionality by combining two inexpensive USB controllers, and
   (c) Lowers the *knowledge requirement* for USB testing by exposing existing host/device communication; no prior knowledge of the USB protocol is required to test (or fuzz) software.
2. **Bug-finding results and analysis**, for which we deliver an interpretation of the effectiveness in finding bugs using this approach.
3. **Enhanced USB testing methods** where we suggest further applications that the TTWE framework affords, such as cataloging USB host/peripheral responses.

As a departure point for discussing the framework architecture, an overview of the USB protocol is provided in Section 2. We proceed by discussing the architecture in depth, and the necessary logic for achieving the testing framework in Section 3. This is followed by our bug-finding results in Section 4 and an analysis of the framework in Section 5. Finally, we discuss related work and conclude in Sections 6 and 7, respectively.

2 Background

This section aims to give a high-level overview of the USB specification so that the framework implementation can be understood. In particular, the USB specification makes use of varying *transfer types*; transfers contain *requests*, *descriptors*, or *data* which are sent over *endpoints* to facilitate communication between a host and peripheral. We explain these in turn before considering the design of the framework.

2.1 Requests and Descriptors

The USB specification defines eleven standard requests. A request from the host, such as `get_descriptor`, expects to receive a descriptor from the peripheral, while other requests such as `set_address` require the peripheral to perform an action. The peripheral responds with descriptors of varying types, containing the respective information. Requests and descriptors allow the USB host to learn about the capabilities of the peripheral, and are also used to load the appropriate software driver.

The USB specification defines eleven standard descriptor types, including device, configuration, and endpoint descriptors. Other descriptors may exist, like the Human Interface Device (HID) descriptor for keyboards and other input devices. A complete listing of requests and descriptors are available in the USB Device Working Group documentation [5].

2.2 Endpoints

USB endpoints provide a way for the host and peripheral to agree on the *direction* and *address* for a given transfer. The address is a number between 0 and 15. Endpoint 0 is a special address; it is a bi-directional endpoint that is used during *control* transfers (see Section 2.3 below). All USB devices must support communication on endpoint 0. Control transfers require bi-directional communication between host and peripheral, and therefore endpoint 0 uses both an `IN` and `OUT` direction. Endpoints for non-control transfers carry data that is particular to the peripheral. These endpoints, numbered 1 through 15, are unidirectional; an endpoint can either support an `IN` or an `OUT` direction, but not both. If bi-directional communication is required outside of endpoint 0, then separate endpoints are needed. For example, a USB mass-storage device may designate in its endpoint descriptor that it will use endpoint 1 with an `IN` direction, and endpoint 2 with an `OUT` direction.

Endpoint *directions* are always specified from the perspective of the *host*. Furthermore, endpoint *addresses* are always defined by the *peripheral's* USB controller.

2.3 Transfer types

Four USB transfer types exist: control, bulk, interrupt, and isochronous. Transfer types are used for different purposes; in this paper, we distinguish simply between control and non-control transfers. Control transfers establish initial communication between a host and peripheral, whereas non-control transfers typically carry data specific to peripheral functionality. For instance, a mass-storage USB device would transfer file data using a bulk transfer, and a keyboard would transfer keystrokes using an interrupt transfer. In both instances, these peripherals would first establish basic communication with control transfers.

2.3.1 Control transfers

Control transfers constitute the means by which a host learns about the peripheral attached to it. Control transfers comprise *requests* and *descriptors* as discussed in Section 2.1. The host *enumerates* the peripheral by sending *requests*, and the peripheral responds either with data (in a *descriptor*) or by performing an appropriate action. Control transfers only take place over endpoint 0. Once *enumeration* over endpoint 0 is complete, the host and peripheral can start exchanging data over non-control endpoints.

Control transfers have additional structure in the form of stages: setup, data, and status stages. Depending on the request, different stages take place. For instance, a peripheral will respond with a descriptor in the data stage in response to a `get_descriptor` request. In a `set_address` request, however, there is no data stage, and the peripheral will only respond with a status stage, indicating whether it was successful (`ACK`), is still busy (`NAK`), or failed (`STALL`). In Section 3.2 we consider how these considerations play a role in the implementation of the framework.

2.3.2 Non-control transfers

Non-control transfers (bulk, interrupt, and isochronous transfers) take place after the host has enumerated the peripheral. In summary, bulk transfers are used for large, time-insensitive data, isochronous for time-critical, real-time data, and interrupt transfers for periodic, low-bandwidth data. On most USB controller chips, bulk and interrupt transfers are treated the same; it is up to the host to signal when data is to be sent or received. Furthermore, non-control transfers may rely on additional, peripheral-specific protocols. One example includes mass-storage devices that commonly use the SCSI (Small Computer System Interface) protocol. Some examples of data that is sent using non-control transfers are provided in the table below.

Peripheral Type	**Action**	**Transfer Type**
Keyboard	Key presses	Interrupt
Mouse	Navigation and clicks	Interrupt
Mass-storage	File operations	Bulk
Printer	File transfer	Bulk
Microphone	Audio recording	Isochronous
Speaker	Audio playback	Isochronous

3 Framework Architecture

We remind the reader that the goal of the framework is to allow one to tap into USB communication between a host and peripheral device in order to *read* and *modify* the data being transferred. In this section we consider a conceptual overview of the framework, proceeded by the hardware and software requirements to achieve effective operation.

3.1 Design

With the aid of two bespoke hardware testing devices, namely, two Facedancer devices [2], we are able to expose USB communication to a Mediating Computer (MC) with a man-in-the-middle strategy. These devices are essentially USB controllers that can act as either a USB host *or* device; detail of their hardware is given in 3.2. On its own, the Facedancer device can perform USB host or device emulation via software driven commands from a computer.

In our design, and with reference to Figure 1, we place one Facedancer in Peripheral Emulation Mode to interact with a USB `HOST`, and a second Facedancer acting in Host Emulation Mode to interact with a USB `PERIPHERAL`. By monitoring the hardware interrupts triggered on the Facedancer USB controllers, the MC is able to mediate communication by forwarding requests from the `HOST`, and responses from the `PERIPHERAL`. The `HOST` is under the impression that it is communicating with an authentic USB peripheral, but in fact it is an *emulated* USB peripheral whose responses are precisely that of the authentic USB `PERIPHERAL` monitored by the Facedancer in Host Emulation Mode. Similarly, the `PERIPHERAL` device is under the impression that it is communicating with an authentic host, whose requests are in fact *emulated* by continuously monitoring the authentic `HOST` using the Facedancer in Peripheral Emulation Mode.

The MC monitors the host by checking whether the Facedancer has received host requests or data, and also forwards device responses to the host via the Facedancer. This operation is performed by a USB client driver on the MC. In likewise manner, a USB Host driver monitors the Facedancer, and forwards host requests. The USB client and host driver exchange data via named pipes on the MC—this allows the USB data to be exposed and manipulated by any intermediary software, such as a mutation fuzzer, before the data continues on its normal course.

Whereas understanding the design is straightforward, there are a number of caveats which present themselves when implementing the framework; we address these in the next section.

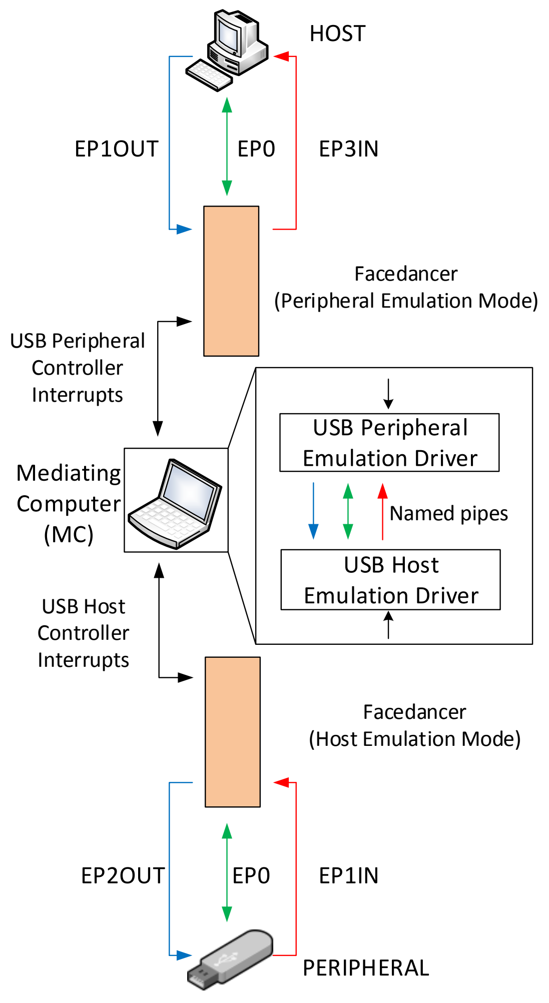
HOST
EP1OUT
EP0
EP3IN
Facedancer
(Peripheral Emulation Mode)
USB Peripheral Controller Interrupts
USB Peripheral Emulation Driver
Named pipes
USB Host Emulation Driver
Mediating Computer (MC)
USB Host Controller Interrupts
Facedancer
(Host Emulation Mode)
EP2OUT
EP0
EP1IN
PERIPHERAL

Figure 1: The TTWE Framework Architecture

3.2 Implementation

The design addresses the conceptual way in which we are able to mediate and tap into USB communication. There are, however, aspects of the USB protocol that do not translate readily to the framework design. These aspects include a) endpoint address numbering, and b) USB handshaking that pertain to control transfers without data stages. In this section we discuss the hardware and software requirements of the framework, and furthermore emphasize the role of software in overcoming the caveats mentioned.

3.2.1 Hardware

The functionality of the Facedancer device is central to the operation of the framework. Currently, the Facedancer is an affordable device [2] with attractive USB testing abilities for the purposes of our framework. However, any device with similar functionality can serve as a substitute; the framework is not specific to the Facedancer. For this reason, we briefly mention the major hardware components that bring about the required functionality.[3]

As shown in Figure 2, the main hardware components of the Facedancer are an FTDI USB/serial adapter chip, a 16-bit microcontroller, and a MAX3421E USB controller chip. USB emulation can be performed by sending software-driven USB data and commands to the microcontroller via the FTDI adapter. The microcontroller drives the USB controller, which may be placed in either *host* or *peripheral* mode by toggling a `mode` bit in one of the controller registers. Recall that in Figure 1, we place one Facedancer's USB controller in host mode, and the other in peripheral mode. The microcontroller listens for responses from the USB controller, which are in turn forwarded back to the computer driving emulation. Basic firmware is required for the microcontroller to perform these actions; such firmware is readily available for the Facedancer components [1].

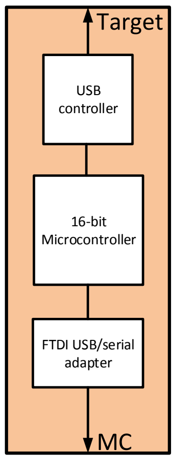

Figure 2: The Facedancer hardware design for performing USB device emulation

3.2.2 Software

The USB host and client drivers monitor the respective USB controller interrupts to determine when data can be sent and received. The drivers process the endpoint source and destination of data once it is available, and this data is sent between the drivers on dedicated named pipes. Another important responsibility of the software is handling special cases of the USB protocol. Problems that arise due to endpoint numbering are addressed by an *endpoint hijacking* approach. Complications in USB handshaking are handled by emulating certain aspects of the handshake procedure.

Endpoint Hijacking When a host receives an endpoint descriptor from a device in response to a `get_configuration` request, it is informed of the endpoint address(es) that the device intends to use for non-control transfers. For example, consider the USB device in Figure 1 (depicted as a mass-storage stick), which requires a bulk `IN` endpoint with address 1, and a bulk `OUT` endpoint with address 2. Recall from Section 2.2 that the direction is always from the perspective of the host, but the endpoint addresses, directions, and transfer types are specified in an endpoint descriptor which are fixed by the peripheral's USB controller,

When the MAX3421E USB controller operates in Host Emulation Mode, it is able to send and receive on *any* endpoint number. However, when the MAX3421E USB controller of the Facedancer in Figure 2 is operated in Peripheral Emulation Mode, its endpoint capabilities are fixed in hardware, and cannot be changed. These capabilities are as follows:

Endpoint Address	**Direction**	**Transfer Type**
EP0	IN/OUT	Control
EP1	OUT	Non-control
EP2	IN	Non-control
EP3	IN	Non-control

Consider that an authentic USB peripheral may have *different* endpoint capabilities, which are also fixed. It is thus possible that endpoint addresses can be mismatched between the emulated peripheral's USB controller and the authentic USB peripheral's. In our case, the USB controller supports only an `OUT` direction on `EP1`, yet the authentic peripheral specifies that it must use `EP1` with an `IN` direction. To overcome this problem, we "hijack" and modify the endpoint descriptor with the MC. When the MC detects an endpoint descriptor being sent from the authentic peripheral in response to a `get_endpoint_descriptor` from the host, it creates a new mapping whereby everything received on `EP1IN` of the USB host emulator will be sent on `EP3IN` of the USB peripheral emulator. Similarly, a mapping is created for the `EP2OUT`/`EP1OUT` pipe.

With the endpoint mapping in place, the authentic host and device can continue communication on the perceived information pipes. Since the USB specification does not mandate the capabilities of endpoints other than endpoint 0, the nature of the endpoint addressing scheme in our framework is thus obscured from the host and device, allowing a transparent data channel. With this scheme we can cope with any manner of endpoint addressing that a peripheral may require, provided the USB controller supports the same number of endpoints.

Emulating Handshaking A further consideration in our framework concerns USB control transfers which do not have a data stage. These control transfers include `set_address`, `set_configuration`, `set_interface`, and `clear_feature` requests. Because these transfers do not have a data stage, the peripheral would simply respond with a status packet, such as an `ACK`. The framework makes provision for mediating data transfers across the pipes, but for status packets, one of two work-around solutions is required:

1. After forwarding the host request, blindly acknowledge it with the peripheral emulator, without knowing the authentic peripheral's status result.

2. Communicate the authentic peripheral's status result once available. This requires extra logic so that custom status messages can be communicated between emulator drivers.

We opted for the former approach. This allows us to asynchronously `ACK` requests, and assume that the request will be successfully processed by the authentic peripheral. This poses the question, what if the request is not processed successfully, and the authentic peripheral responds with something other than an `ACK`? In such an event, subsequent requests from the host would not receive a response, and the breakdown of communication would become apparent on the host and client driver software. During testing, we observed this communication breakdown when we neglected to blindly `ACK` the aforementioned requests. After doing so, we did not encounter the scenario again in obtaining the results of Section 4, so the approach proved sufficient for such purposes.

4 Results

In this section we discuss the preliminary results of our framework in exposing USB communication, as well as USB testing and fuzzing. The first significant result is the ability to expose USB communication between an authentic host and device, enabling man-in-the-middle attacks without requiring prior knowledge of the USB protocol. The second significant result is the discovery of new bugs in USB software. Due to the recent implementation of our USB framework, a limited time of approximately one week was spent fuzzing various hosts. Despite this brief period of dedicated testing, we discovered security-relevant bugs. As vulnerability disclosure is still being coordinated at the time of writing, critical details have been omitted.

4.1 USB Enumeration & Functionality

Device Enumeration We have been able to successfully mediate communication between host and peripheral pairs, allowing the host to fully enumerate a variety of USB peripheral classes. These include the USB HID, mass-storage, printer, and imaging device classes. Because of the generic manner in which enumeration is performed, we expect most device classes to work with this framework in a plug-and-play manner.

Device Functionality USB mass-storage functionality can be achieved successfully after device enumeration takes place. This means that we can perform mount, browse, read, and write actions on the mass-storage device from the host while tapping into USB communication. Consider Table 1 which presents mass-storage functionality by Two-Way Emulation on a Linux host. The orange row indicates the peripheral's endpoint descriptor, sent in the `IN` direction (`DIR`) on endpoint (`EP`) 0. The bold values **1** and **130** are the endpoint addresses that are to be hijacked and modified on the fly. The endpoint address corresponds to the lower nibble of the hexadecimal representation of these values: `1h` and `2h` respectively, with direction specified in the most significant bit (`0h`, or `OUT`, and `1h`, or `IN` respectively). The subsequent yellow rows indicate the start of SCSI data transmitting on the desired endpoints `EP1OUT` and `EP3IN` after enumeration. Finally, the green row indicates a `set_configuration` request by the host in the `OUT` direction; here, the host would receive a blind `ACK`, and the request is forwarded to the mass-storage device.

DIR	EP	DATA (Base 10)
OUT	[0]	[128, 6, 0, 1, 0, 0, 64, 0]
IN	[0]	[18, 1, 0, 2, 0, 0, 0, 64, 143, 5, 135, 99, 2, 1, 1, 2, 3, 1]
OUT	[0]	[0, 5, 25, 0, 0, 0, 0, 0]
OUT	[0]	[128, 6, 0, 1, 0, 0, 18, 0]
IN	[0]	[18, 1, 0, 2, 0, 0, 0, 64, 143, 5, 135, 99, 2, 1, 1, 2, 3, 1]
OUT	[0]	[128, 6, 0, 6, 0, 0, 10, 0]
IN	[0]	[10, 6, 0, 2, 0, 0, 0, 64, 1, 0]
OUT	[0]	[128, 6, 0, 2, 0, 0, 9, 0]
IN	[0]	[9, 2, 32, 0, 1, 1, 0, 128, 100]
OUT	[0]	[128, 6, 0, 2, 0, 0, 32, 0]
IN	[0]	[9, 2, 32, 0, 1, 1, 0, 128, 100, 9, 4, 0, 0, 2, 8, 6, 80, 0, 7, 5, **1**, 2, 64, 0, 0, 7, 5, **130**, 2, 64, 0, 0]
OUT	[0]	[128, 6, 0, 3, 0, 0, 255, 0]
IN	[0]	[4, 3, 9, 4]
OUT	[0]	[128, 6, 2, 3, 9, 4, 255, 0]
IN	[0]	[26, 3, 77, 0, 97, 0, 115, 0, 115, 0, 32, 0, 83, 0, 116, 0, 111, 0, 114, 0, 97, 0, 103, 0, 101, 0]
OUT	[0]	[128, 6, 1, 3, 9, 4, 255, 0]
IN	[0]	[16, 3, 71, 0, 101, 0, 110, 0, 101, 0, 114, 0, 105, 0, 99, 0]
OUT	[0]	[128, 6, 3, 3, 9, 4, 255, 0]
IN	[0]	[18, 3, 49, 0, 57, 0, 54, 0, 50, 0, 51, 0, 55, 0, 51, 0, 54, 0]
OUT	[0]	[0, 9, 1, 0, 0, 0, 0, 0]
OUT	[0]	[161, 254, 0, 0, 0, 0, 1, 0]
IN	[0]	[0]
OUT	[1]	[85, 83, 66, 67, 1, 0, 0, 0, 36, 0, 0, 0, 128, 0, 6, 18, 0, 0, 0, 36, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
IN	[3]	[0, 128, 4, 2, 31, 0, 0, 0, 71, 101, 110, 101, 114, 105, 99, 32, 70, 108, 97, 115, 104, 32, 68, 105, 115, 107, 32, 32, 32, 32, 32, 32, 56, 46, 48, 55]
IN	[3]	[85, 83, 66, 83, 1, 0, 0, 0, 0, 0, 0, 0, 0]
OUT	[1]	[85, 83, 66, 67, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
IN	[3]	[85, 83, 66, 83, 2, 0, 0, 0, 0, 0, 0, 0, 1]
OUT	[1]	[85, 83, 66, 67, 3, 0, 0, 0, 18, 0, 0, 0, 128, 0, 6, 3, 0, 0, 0, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
…	…	…

Table 1: Mass-storage device functionality by TTWE on a Linux Host

Other communication in this table corresponds to standard USB enumeration behavior. We were able to achieve functionality in this fashion by writing additional software for the Facedancer that handles, for example, the granularity of bulk transfers in 512-byte blocks. If we wanted to support HID functionality, it would require writing additional software that simply monitors the interrupt endpoints on the Facedancer. Thus, achieving device functionality after enumeration may require additional code, but fortunately this is a software restriction, not a hardware one. Furthermore, this additional implementation in software pertains only to endpoint types, and is not device or vendor specific.

4.2 USB Fuzzing

USB Printer Driver We discovered a bug in a popular operating system's core USB printer driver, which causes a kernel panic and system crash. Analysis of the crash dump revealed that this is due to an arbitrary memory read, resulting after heap corruption on the host. At present this bug causes a denial-of-service, but it may also allow arbitrary code execution. The bug was found by rudimentary fuzzing techniques—bytes were randomly mutated in flight during USB device enumeration. Initially, the bug was found by using a real printer; this was important since the host issued different requests depending on the printer's malformed responses. In particular, the bug is only triggered in response to a host request for a specific string descriptor. After inducing a system crash on the host, we could analyze the printer response and replay it, this time using only peripheral emulation.

USB Host Printer denial-of-service Another bug was discovered in a popular operating system which would cause an application to hang as soon as the user attempts to print something. This was not specific to any application, and would occur in word-processing programs, browsers, editors, and so forth. We could thus envision a scenario where an attacker may use a malicious USB device to perform a denial-of-service attack on a locally accessible computer. As long as the device remained in the computer, the user would need to forcibly terminate the application from which they attempted to print. It was found that this bug was a result of the host waiting in a busy-loop for certain USB `ACK` responses from the peripheral. Again, this bug was found with the aid of a real printer. After discovering the cause, we could reproduce the attack by replaying it using pure emulation. As before, the bug was triggered by refraining from responding to a specific host request.

USB Host Printer denial-of-service A certain printer which supports printing from a USB mass-storage device was found to contain a bug which causes the printer to crash. By using a real USB mass-storage device and fuzzing the printer host, the bug is triggered by a malformed SCSI response. As with previous bugs, we were able to replay the responses so that the printer could be crashed at will.

5 Analysis

Preliminary results indicate that our framework is effective, having detected novel bugs that would otherwise require much more effort to find using traditional emulation techniques. There are a number of improvements that the framework delivers over traditional solutions.

Flexibility While the framework allows platform-independent man-in-the-middle attacks on the USB protocol, this important feature is also possible with a USB analyzer [4]. However, Davis [9] mentions the limitations of the software provided with the tools: they lack a proper software API, requiring the user to make use of a custom scripting language [10]. Our framework does not place restrictions on the user's ability to manipulate USB data. The host and client emulation drivers are implemented in Python and expose the raw USB data over Unix pipes. If desired, other forms of inter-process communication (IPC) such as Unix sockets could also be employed. We opted for named pipes due to the simplicity of having host and peripheral emulation drivers attach to these endpoints, instead of implementing extra IPC functionality in the drivers themselves. Moreover, the user is not restricted to Python, and if desired, may implement drivers that interact with the Facedancer in any preferred language. A further advantage of our framework is the ability to immediately replay USB communication from either a host or peripheral in an emulated fashion—the user need not write additional scripts to generate USB traffic.

Cost Effective Despite the software limitation of the USB analyzer, its capabilities go beyond that of other tools and pure software solutions. Davis [11] attested in 2013 that it is the preferred device for finding USB bugs, although at a cost of approximately $1,400. In contrast, the hardware required by the Facedancer's in our solution would cost approximately $150 [3].

Background Knowledge Requirement As mentioned, a USB analyzer would typically require understanding of the USB protocol for generating USB traffic, as well as knowledge of a custom scripting language. Similarly, a single bespoke testing device such as the Facedancer requires knowledge of the USB protocol so that it can be programmed to respond appropriately to requests. For rudimentary fuzzing purposes, our solution requires very little knowledge about the underlying protocol, since the Facedancers simply relay device responses. By exposing the raw communication, users are able to capture, modify, and replay data between host and device.

Limitations There is a notable delay in USB communication between the host and peripheral due to the emulation devices. Currently, the average delay between USB control transfers is 300 milliseconds, granting enough time to perform USB enumeration in our setup. However, it was found that general mass-storage actions, as discussed in Section 4, take an extended amount of time. For example, it may take a few minutes to mount a mass-storage device, or list its contents. This delayed behavior can therefore impact the speed of fuzzing throughput on certain aspects of USB code.

One way to address the delay in communication may be to optimize the interrupt handling code in the Facedancer client and host drivers. Currently, the authentic peripheral is polled until a response is received, which is then forwarded to the host. Some hosts have a shorter timeout for peripheral responses, and may reissue redundant requests if the data is not relayed when it's immediately available; fine-grained tuning of poll intervals could thus quicken transmission. This could also be explored in tandem with communicating the entire handshaking process instead of performing blind `ACK`s for some requests, as mentioned in Section 3.2.2.

6 Related Work

The Facedancer [2] presents recent advances in the arena of USB software testing by way of device emulation techniques. Bratus et al. demonstrate how this technique enables "efficient injection of arbitrary traffic" into the USB bus, while being affordable [8].

Software-only techniques for USB testing have been considered, where USB devices are emulated in a guest operating system environment [6]. Similar fuzzing in various virtualized OS environments by Jodeit et al. [12] has triggered bugs in virtualization software and USB drivers. Although this technique has proven its ability to find bugs, its effectiveness is limited to the capabilities of the virtualization software. According to Jodeit et al., a further limitation includes the difficulty in reproducing crashes.

Davis's use of USB analyzers has produced a number of CVE (Common Vulnerabilities and Exposures) identifiers relating to USB security on Windows, Solaris, and OS X [9, 11]. Davis also demonstrates the Frisbee USB testing automation by using USB analyzers within the confines of a cumbersome scripting language [10]. Still, this has proven to be one of the best approaches to date.

7 Conclusion

In this work we presented a framework that exposes USB communication between a host and peripheral in a flexible, cost-effective way. The framework contributes to the growing demands of USB testing software by enabling man-in-the-middle attacks, and allowing users to fuzz host and device USB software. We detected novel bugs that resulted by modifying USB data during real-time communication, and found that the bugs can be reproduced by replaying the captured responses in an emulated fashion.

We believe there are a number of additional applications for the TTWE framework beyond those covered in this paper. For one, the framework could be used to capture authentic host/peripheral USB communication which may be catalogued to serve as seed values for randomized fuzzing. This ability could speed up fuzzing, and in some instances substitute for the need of a physical device. Secondly, we envision that the framework can be leveraged in exploitation techniques such as Mulliner's mass-storage TOCTTOU attack [13]. In lieu of maintaining two emulated mass-storage filesystems as per the original approach, and having to "switch between the original and modified filesystem image" in order to exploit the "read-it-twice" condition, one could instead modify the data on the fly from a single filesystem image at the exact moment that modification is desired. Such an ability is made evident from the discussion in Section 4.1.

The prospects of vulnerability discovery and exploitation of the USB attack surface are expected to increase as the tools for doing so improve. To echo the words of Joshua Wright, "Security will not get better until tools for practical exploration of the attack surface are made available" [15]. We believe that the TTWE framework makes practical exploration of the USB attack surface flexible and affordable, thereby contributing to the effort of securing USB software.

8 Acknowledgements

A hearty thank you to Willem Bester for taking the time outside of his regular duties to deliver insightful comments during the review of this work. The authors also wish to thank Naspers for their financial support during the course of this research.

References

[1] Facedancer Firmware Source Repository. `https://goodfet.svn.sourceforge.net/svnroot/goodfet`. Accessed 25/05/2014.

[2] GoodFET Facedancer. `https://goodfet.sourceforge.net/hardware/facedancer21/`. Accessed 25/05/2014.

[3] int3cc. `http://int3.cc/products/facedancer21`. Accessed 25/05/2014.

[4] MQP Electronics. `http://www.mqp.com/usb500.htm`. Accessed 25/05/2014.

[5] USB Device Working Group. `http://www.usb.org/developers/docs/devclass_docs/`. Accessed 25/05/2014.

[6] MWR InfoSecurity. `https://labs.mwrinfosecurity.com/blog/2011/07/14/usb-fuzzing-for-the-masses/`, 2011. Accessed 25/05/2014.

[7] BARRALL, D., AND DEWEY, D. Plug and root, the USB key to the kingdom. In *Black Hat Briefings* (jul. 2005).

[8] BRATUS, S., GOODSPEED, T., JOHNSON, P. C., SMITH, S. W., AND SPEERS, R. Perimeter-crossing buses: A new attack surface for embedded systems. In *8th Workshop on Embedded Systems Security (WESS)* (sept. 2013).

[9] DAVIS, A. USB - Undermining security barriers. In *Black Hat Briefings* (aug. 2011).

[10] DAVIS, A. Fuzzing USB devices using frisbeelite. Tech. rep., NGS Secure Research, jan. 2012.

[11] DAVIS, A. USB driver vulnerabilities whitepaper. Tech. rep., NCC Group, jan. 2013.

[12] JODEIT, M., AND JOHNS, M. USB device drivers: A stepping stone into your kernel. In *2010 European Conference on Computer Network Defense* (2010).

[13] MULLINER, C., AND MICHELE, B. Read it twice! A mass-storage-based tocttou attack. In *6th USENIX Workshop on Offensive Technologies* (aug. 2012).

[14] VEGA, R. D. Linux kernel USB device driver - buffer overflow. Tech. rep., MWR InfoSecurity, oct. 2009.

[15] WRIGHT, J. `http://code.google.com/p/zigbee-security/`. Accessed 25/05/2014.

Notes

[1] CVE advisories related to USB software bugs

2013	2012	2011	2010	2009
CVE-2013-1285	CVE-2012-2693	CVE-2011-0712	CVE-2010-4656	CVE-2009-4067
CVE-2013-1286	CVE-2012-3723	CVE-2011-2295		
CVE-2013-1287				
CVE-2013-2888				
CVE-2013-2889				
CVE-2013-2890				
CVE-2013-2891				
CVE-2013-2892				
CVE-2013-2893				
CVE-2013-2894				
CVE-2013-2895				
CVE-2013-2896				
CVE-2013-2897				
CVE-2013-2898				
CVE-2013-2899				
CVE-2013-3200				

[2] $75 at the time of writing

[3] A complete bill of materials for the Facedancer may be found at http://goodfet.sourceforge.net/hardware/facedancer21/